INF4140 - Models of concurrency

Fall 2016

November 4, 2016

Abstract

This is the "handout" version of the slides for the lecture (i.e., it's a rendering of the content of the slides in a way that does not waste so much paper when printing out). The material is found in [Andrews, 2000]. Being a handout-version of the slides, some figures and graph overlays may not be rendered in full detail, I remove most of the overlays, especially the long ones, because they don't make sense much on a handout/paper. Scroll through the real slides instead, if one needs the overlays.

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture.

1 Message passing and channels

17. Oct. 2016

1.1 Intro

Outline

Course overview:

- Part I: concurrent programming; programming with shared variables
- Part II: "distributed" programming

Outline: asynchronous and synchronous message passing

- Concurrent vs. distributed programming[1]
- Asynchronous message passing: channels, messages, primitives
- Example: filters and sorting networks
- From monitors to client–server applications
- Comparison of message passing and monitors
- About synchronous message passing

Shared memory vs. distributed memory

more traditional system architectures have one shared memory:

- many processors access the same physical memory
- example: fileserver with many processors on one motherboard

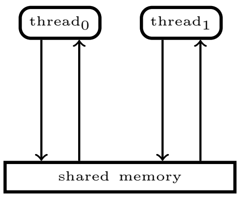
Distributed memory architectures:

- Processor has private memory and communicates over a "network" (inter-connect)
- Examples:
  - Multicomputer: asynchronous multi-processor with distributed memory (typically contained inside one case)

[1]The dividing line is not absolute. One can make perfectly good use of channels and message passing also in a non-distributed setting.

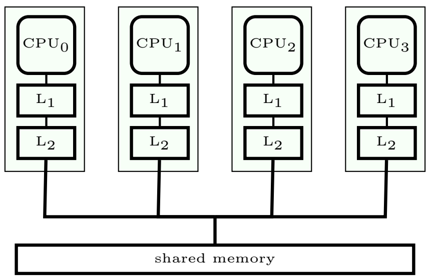
  - Workstation clusters: PC's in a local network
  - Grid system: machines on the Internet, resource sharing
  - cloud computing: cloud storage service
  - NUMA-architectures
  - cluster computing ...

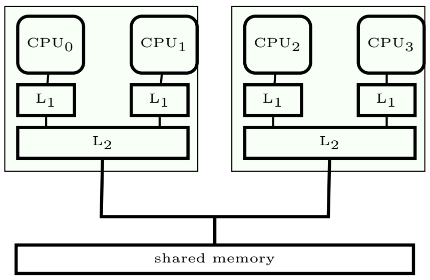
Shared memory concurrency in the real world

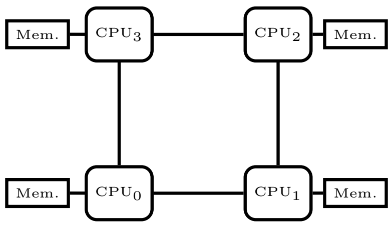


- the memory architecture does not reflect reality
- out-of-order executions:
  - modern systems: complex memory hierarchies, caches, buffers...
  - compiler optimizations,

SMP, multi-core architecture, and NUMA







Concurrent vs. distributed programming

Concurrent programming:

- Processors share one memory
- Processors communicate via reading and writing of shared variables

Distributed programming:

- Memory is distributed $\Rightarrow$ processes cannot share variables (directly)
- Processes communicate by sending and receiving *messages* via shared *channels*

  or (in future lectures): communication via *RPC* and *rendezvous*

1.2 Asynch. message passing

Asynchronous message passing: channel abstraction

Channel: abstraction, e.g., of a physical communication network[2]

- One–way from sender(s) to receiver(s)
- unbounded FIFO (queue) of waiting messages
- preserves message order
- atomic access
- error–free
- typed

Variants: errors possible, untyped, ...

Asynchronous message passing: primitives

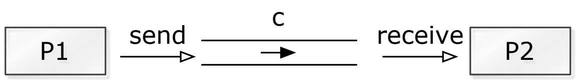
Channel declaration

$$\text{chan c}(\text{type}_1\text{id}_1, \ldots, \text{type}_n\text{id}_n);$$

Messages: n-tuples of values of the respective types

communication primitives:

- send $c(\text{expr}_1, \ldots, \text{expr}_n)$; Non-blocking, i.e. asynchronous
- receive $c(\text{var}_1, \ldots, \text{var}_n)$; Blocking: receiver waits until message is sent on the channel
- empty (c); True if channel is empty



Simple channel example in Go

```
func main() {
        messages := make(chan string, 0)  // declare + initialize

        go func() { messages <- "ping" }() // send
        msg := <-messages                  // receive
        fmt.Println(msg)
}
```

Short intro to the Go programming language

- programming language, executable, used by f.ex. Google
- supporting channels and asynchronous processes (function calls)
  - *go-routine*: a lightweight thread
- syntax: mix of functional language (lambda calculus) and imperative style programming (built on C).

[2]but remember also: producer-consumer problem

Some syntax details of the Go programming language

Calls

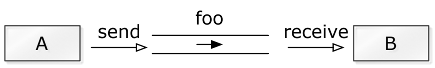
- $f(x)$ – ordinary (synchronous) function call, where f is a defined function or a functional definition
- **go** $f(x)$ – called as an asynchronous process, i.e. go-routine **Note**: the go-routine will die when its parent process dies!
- **defer** $f(x)$ – the call is delayed until the end of the process

Channels

- $chan := make(chanint, buffersize)$ – declare channel
- $chan < -x$ – send x
- $< -chan$ – receive
- example: $y :=< -chan$ – receive in y

Run command: `go run program.go` – compile and run program

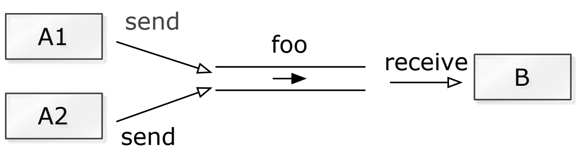
Example: message passing



(x,y) = (1,2)

```
chan foo(int);

process A {
  send foo(1);
  send foo(2);
}

process B {
  receive foo(x);
  receive foo(y);
}
```

Example: shared channel



(x,y) = (1,2) or (2,1)

```
process A1 {
  send foo(1);
}

process A2 {
  send foo(2);
}

process B {
  receive foo(x);
  receive foo(y);
}
```

```
func main() {
        foo  := make(chan int, 10)
        go func() {
                time.Sleep(1000)
                foo  <- 1        // send
        }()

        go func() {
                time.Sleep(1)
                foo  <- 2
        }()
        fmt.Println("first␣=␣", <-foo)
        fmt.Println("second␣=␣", <-foo)
}
```

Asynchronous message passing and semaphores

Comparison with general semaphores:

channel	$\simeq$	semaphore
send	$\simeq$	**V**
receive	$\simeq$	**P**

Number of messages in queue = value of semaphore

(Ignores content of messages)

Semaphores as channels in Go

```
type dummy interface {}              // dummy type,
type Semaphore chan dummy            // type definition

func  (s Semaphore) Vn (n int) {
        for i:=0; i<n; i++ {
                s <- true            // send something
        }
}
func  (s Semaphore) Pn (n int) {
        for i:=0; i<n; i++ {
                <- s                 // receive
        }
}

func  (s Semaphore) V () {
        s.Vn(1)
}
func  (s Semaphore) P () {
        s.Pn(1)
}
```

Listing 1: 5 Phils

```
package main
import ("fmt"
        "time"
        "sync"
        "math/rand"
        "andrewsbook/semchans")      // semaphores using channels


var wg sync.WaitGroup

const m = 5                  // let's make just 5
var forks  = [m]semchans.Semaphore {
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1),
        make (semchans.Semaphore,1)}
```

```
func main () {
        for i:=0; i<m; i++ {    // initialize the sem's
                forks[i].V()
        }
        wg.Add(m)
        for i:=0; i<m; i++ {
                go philosopher(i)
        }
        wg.Wait()

}

func philosopher(i int) {
  defer wg.Done()
  r := rand.New(rand.NewSource(99))    // random generator
  fmt.Printf("start␣P(%d)\n",i)
  for true {
    fmt.Printf("P(%d)␣is␣thinking\n",i)
    forks[i].P()
//    time.Sleep(time.Duration(r.Int31n(0)))  // small delay for DL
    forks[(i+1)%m].P()
    fmt.Printf("P(%d)␣starts␣eating\n",i)
    time.Sleep(time.Duration(r.Int31n(5)))  // small delay
    fmt.Printf("P(%d)␣finishes␣eating\n",i)
    forks[i].V()
    forks[(i+1)%m].V()
  }
}
```

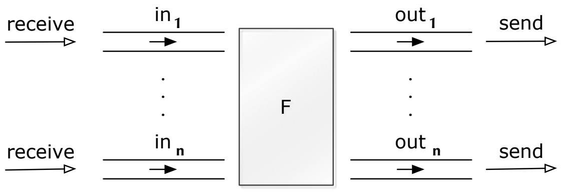
1.2.1 Filters

Filters: one–way interaction

Filter F

= process which:

- receives messages on input channels,
- sends messages on output channels, and
- output is a function of the input (and the initial state).



- A filter is specified as a predicate.
- Some computations: naturally seen as a composition of filters.
- cf. *stream* processing/programming (feedback loops) and *dataflow programming*

Example: A single filter process

Problem: Sort a list of n numbers into ascending order.

process **Sort** with input channels **input** and output channel **output**.

Define:

n : number of values sent to **output**. $sent[i]$: i'th value sent to **output**.

Sort predicate

$\forall i : 1 \leq i < n. \big(sent[i] \leq sent[i+1]\big) \;\wedge$ values sent to **output** are a *permutation* of values from **input**.

Filter for merging of streams

Problem: Merge two sorted input streams into one sorted stream.

Process Merge with input channels $\mathbf{in}_1$ and $\mathbf{in}_2$ and output channel **out**:

```
  in₁: 1 4 9 ...
                              out: 1 2 4 5 8 9 ...
  in₂: 2 5 8 ...
```

Special value **EOS** marks the end of a stream.

Define: n : number of values sent to **out**. $sent[i]$: i'th value sent to **out**.

The following shall hold when **Merge** *terminates*:

$\mathbf{in}_1$ and $\mathbf{in}_2$ are empty $\wedge$ $sent[n+1] = \mathbf{EOS} \wedge \quad \forall i : 1 \leq i < n\big(sent[i] \leq sent[i+1]\big) \wedge$ values sent to **out** are a *permutation* of values from $\mathbf{in}_1$ and $\mathbf{in}_2$

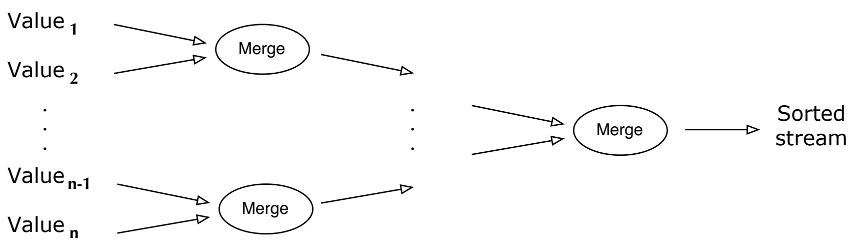
Example: Merge process

```
chan in1(int), in2(int), out(int);

process Merge {
  int v1, v2;
  receive in1(v1);            # read the first two
  receive in2(v2);            # input values

  while (v1 ≠ EOS and v2 ≠ EOS) {
    if (v1 ≤ v2)
      { send out(v1); receive in1(v1); }
    else                          # (v1 > v2)
      { send out(v2); receive in2(v2); }
  }

                          # consume the rest
                          # of the non-empty input channel
  while (v2 ≠ EOS)
    { send out(v2); receive in2(v2); }
  while (v1 ≠ EOS)
    { send out(v1); receive in1(v1); }
  send out(EOS);    # add special value to out
}
```

Sorting network

We now build a network that sorts n numbers.

We use a collection of Merge processes with tables of shared input and output channels.



(Assume: number of input values n is a power of 2)

1.2.2 Client-servers

Client-server applications using messages

Server: process, repeatedly handling requests from client processes.

Goal: Programming client and server systems with asynchronous message passing.

```
chan request(int clientID, ...),
     reply[n](...);

client nr. i                        server
                                    int id;  # client id.

                                    while(true) {  # server loop
send request(i,args);      ⟶        receive request(id,vars);
:                          :            :
receive reply[i](vars);    ⟵        send reply[id](results);
                                        }
```

1.2.3 Monitors

Monitor implemented using message passing

Classical monitor:

- controlled *access* to shared resource
- Permanent variables (monitor variables): safeguard the resource state
- access to a resource via *procedures*
- procedures: executed under *mutual exclusion*
- *condition* variables for synchronization

also implementable by server process + message passing
Called "active monitor" in the book: active process (loop), instead of passive procedures.[3]

Allocator for multiple–unit resources

Multiple–unit resource: a resource consisting of multiple units

Examples: memory blocks, file blocks.
Users (clients) need resources, use them, and return them to the allocator ("free" the resources).

- here simplification: users get and free *one* resource at a time.
- two versions:
  1. monitor
  2. server and client processes, message passing

Allocator as monitor

Uses "passing the condition" pattern ⇒ simplifies later translation to a server process

Unallocated (free) units are represented as a set, type **set**, with operations **insert** and **remove**.

Recap: "semaphore monitor" with "passing the condition"

```
monitor Semaphore       { # monitor invariant: s ≥ 0
  int s := 0;             # value of the semaphore
  cond pos;               # wait condition

  procedure Psem() {
    if   (s=0)
         wait (pos);
    else
         s := s - 1
  }


  procedure Vsem() {
    if   empty(pos)
           s := s + 1
    else
           signal(pos);
  }
}
```

(Fig. 5.3 in Andrews [Andrews, 2000])

[3]In practice: server may spawn local threads, one per request.

Allocator as a monitor

```
monitor Resource_Allocator {
  int avail := MAXUNITS;
  set units := ... # initial values;
  cond free;         # signalled when process wants a unit

  procedure acquire(int &id) {  # var.parameter
    if (avail = 0)
      wait(free);
    else
      avail := avail-1;
    remove(units, id);
  }

  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail := avail+1;
    else
      signal(free);              # passing the condition
  }
}
```

([Andrews, 2000, Fig. 7.6])

Allocator as a server process: code design

1. interface and "data structure"
   (a) allocator with two types of operations: get unit, free unit
   (b) 1 request channel[4] ⇒ must be *encoded* in the arguments to a request.
2. control structure: nested **if**-statement (2 levels):
   (a) first checks type operation,
   (b) proceeds correspondingly to monitor-**if**.
3. synchronization, scheduling, and mutex
   (a) cannot wait (**wait(free)**) when no unit is free.
   (b) must save the request and return to it later
       ⇒ queue of pending requests (**queue**; **insert**, **remove**).
   (c) request: "synchronous/blocking" call ⇒ "ack"-message back
   (d) no internal parallelism ⇒ mutex

1>In order to design a monitor, we may follow the following 3 "design steps" to make it more systematic: 1) Inteface, 2) "business logic" 3) sync./coordination

Channel declarations:

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitID);
chan reply[n](int unitID);
```

Allocator: client processes

```
process Client[i = 0 to n-1] {
  int unitID;
  send request(i, ACQUIRE, 0)        # make request
  receive reply[i](unitID);          # works as ``if synchronous''
  ...                                # use resource unitID
  send request(i, RELEASE, unitID);  # free resource
  ...
}
```

(Fig. 7.7(b) in Andrews)

[4]Alternatives exist

Allocator: server process

```
process Resource_Allocator {
  int avail := MAXUNITS;
  set units := ...              # initial value
  queue pending;                # initially empty
  int clientID, unitID; op_kind kind; ...
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind = ACQUIRE) {
      if (avail = 0)            # save request
        insert(pending, clientID);
      else { # perform request now
          avail:= avail-1;
          remove(units, unitID);
          send reply[clientID](unitID);
      }
    }
    else {                      # kind = RELEASE
      if empty(pending) {       # return units
        avail := avail+1; insert(units, unitID);
      } else {                  # allocates to waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
} } } }                         # Fig. 7.7 in Andrews (rewritten)
```

Duality: monitors, message passing

monitor-based programs	*message-based programs*
monitor variables	local server variables
process-IDs	**request** channel, operation types
procedure call	**send request**(), **receive reply[i]**()
go into a monitor	**receive request**()
procedure return	**send reply[i]**()
wait statement	save pending requests in a queue
signal statement	get and process pending request (`reply`)
procedure body	branches in **if** statement wrt. op. type

1.3 Synchronous message passing

Synchronous message passing

Primitives:

- New primitive for sending:

  synch_send c($\text{expr}_1, \ldots, \text{expr}_n$);

  Blocking send:

  - sender waits until message is received by channel,
  - i.e. sender and receiver "synchronize" sending and receiving of message

- Otherwise: like asynchronous message passing:

  receive c($\text{var}_1, \ldots, \text{var}_n$);

  empty(c);

Synchronous message passing: discussion

Advantages:

- Gives maximum size of channel.

  Sender synchronises with receiver ⇒ receiver has at most 1 pending message per channel per sender ⇒ sender has at most 1 unsent message

Disadvantages:

- reduced parallellism: when 2 processes communicate, 1 is always blocked.
- higher risk of deadlock.

Example: blocking with synchronous message passing

```
chan values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    ... # computation ...;
    synch_send values(data[i]);
} }

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    ... # computation ...;
} }
```

Assume both producer and consumer vary in time complexity.
Communication using `synch_send`/`receive` will block.

With *asynchronous* message passing, the waiting is reduced.

Example: deadlock using synchronous message passing

```
chan in1(int), in2(int);

process P1 {
  int v1 = 1, v2;
  synch_send in2(v1);
  receive in1(v2);
}

process P2 {
  int v1, v2 = 2;
  synch_send in1(v2);
  receive in2(v1);
}
```

P1 and **P2** block on **synch_send** – deadlock.
One process must be modified to do **receive** first
⇒ asymmetric solution.

With asynchronous message passing (**send**) all goes well.

```
func main() {
        var wg sync.WaitGroup // wait group
        c1,c2  := make(chan int, 0),make(chan int, 0)
        wg.Add(2)              // prepare barrier
        go func() {
                defer wg.Done() // signal to barrier
                c1  <- 1        // send
                x := <- c2      // receive
                fmt.Printf("P1: x := %v\n", x)
        }()

        go func() {
                defer wg.Done()
                c2  <- 2
                x := <- c1
                fmt.Printf("P2: x := %v\n", x)
        }()
        wg.Wait()          // barrier
}
```

References

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley.